A Component-Level Path Composition Approach for Efficient Transient Analysis of Large CTMCs

Vinh V. Lam* Peter Buchholz† William H. Sanders*

*Dept. of Electrical and Computer Engineering and Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 W. Main St., Urbana, IL, U.S.A.
Email: {lam, whs}@crhc.uiuc.edu

†Informatik IV, Universität Dortmund
D-44221 Dortmund, Germany
Email: peter.buchholz@udo.edu

Abstract

Path-based techniques make the analysis of very large Markov models feasible by trading off high computational complexity for low space complexity. Often, a drawback in these techniques is that they have to evaluate many paths in order to compute reasonably tight bounds on the exact solutions of the models. In this paper, we present a path composition algorithm to speed up path evaluation significantly. It works by quickly composing subpaths that are precomputed locally at the component level. The algorithm is computationally efficient since individual subpaths are precomputed only once, and the results are reused many times in the computation of all composed paths. To the best of our knowledge, this work is the first to propose the idea of path composition for the analysis of Markov models. A practical implementation of the algorithm makes it feasible to solve even larger models, since it helps not only in evaluating more paths faster but also in computing long paths efficiently by composing them from short ones. In addition to presenting the algorithm, we demonstrate its application and evaluate its performance in computing the reliability and availability of a large distributed information service system in the presence of fault propagation and in computing the probabilities of buffer overflow and buffer flushing in a media multicast system with varying system configurations.

1. Introduction

Due to ever-increasing size and complexity in system designs, model-based evaluation has become a cost-effective way to study alternative designs before the actual systems are built. Model-based evaluation can be used to estimate the reliability, availability, or performability of the systems, for instance. Often models are used that can be mapped onto continuous-time Markov chains (CTMCs) for solution. It is well-known that as models grow, the sizes of their state spaces grow at an exponential rate. That growth rate can quickly overwhelm the storage capacity of modern computing systems. Thus, new techniques that are effective at managing storage complexity are needed for analyzing large Markov models.

When transient results are required, several analysis methods are available. They can be roughly classified as either state-based or non-state-based and as either approximate or exact. Modern state-based techniques, such as [4, 1, 5, 7], are very effective in representing the state spaces and the corresponding transition matrices compactly, but they are limited by the fact that they must explicitly hold one or more solution or iteration vectors in memory. Since the sizes of those vectors are on the order of the state spaces, the techniques are restricted to solving small models. Consequently, non-state-based techniques like simulation have been generally used to solve large models. Simulation, however, belongs to the class of approximate techniques, since its results are statistical in nature.

An alternative to the previously mentioned techniques is the approach of path-based analysis (e.g., [9, 10, 6]), whereby bounded solutions are computed for large models. Like simulation, path-based techniques derive results for a model by evaluating trajectories over which the corresponding system may proceed over time. In contrast to the approach used in simulation, however, paths are selected in a systematic way. That usually allows one to compute upper and lower bounds, instead of approximations, for the desired measures. Despite their success, existing path-based techniques find very limited application, because they suffer from poor performance. The performance of all path-based techniques is significantly dependent on two factors: (1) the number of paths they can compute in a given amount of time and (2) the relevance of the paths that are computed.

In this paper, we introduce a novel technique for computing paths that improves performance substantially. The technique is based on the algorithm in which paths are composed from subpaths that are precomputed locally for the individual model components. The path composition algorithm helps to eliminate much of the redundant computation so that many more paths can be computed in a given amount of time. Furthermore, we augment the technique with a path-selection algorithm to choose the more relevant paths for analysis. To the best of our knowledge, this work is the first to propose the idea of path composition to improve the performance of path-based techniques for the analysis of Markov models. Our implementation of the algorithm effectively exploits the existence of common subpaths across all paths, resulting in a speedup of 6.6 to 8.8 times in our benchmark models.

The presentation of our paper proceeds as follows. In Section 2, we briefly review the necessary background material on the structured path-based approach. Here we focus specifically on the calculation of rewards and the algorithm for exploration at the path level. Then, in Section 3, we turn our attention to the subpath level and describe the new algorithm for exploring, computing, and composing subpaths. Section 4 presents experimental results for the new algorithm and compares them to those in our previous work. Finally, in Section 5, we conclude with a summary of our current work.

2. Review of the Structured Path-based Approach

In this section, we review the notations that are used throughout this paper, describe what paths are, and show how paths are explored and how reward is calculated on a path basis. A more detailed description and derivation can be found in [6]. This section lays the foundation for the derivation of the path-composition algorithm in the next section.

2.1. Computation of Path Reward and Solution Bounds

We consider a class of models whose infinitesimal generators and solution vectors can be composed from the component matrices and vectors using the Kronecker sum and product operators (see [6] for details). Specifically, the generator matrix $\mathbf{Q}$ (a submatrix of $\hat{\mathbf{Q}}$) of a model in the class can be represented by the composition of component matrices (which is typically used in the literature, e.g., see [11]):

$$\hat{\mathbf{Q}} = \bigoplus_{i=1}^{J} \mathbf{Q}_l^{(i)} + \sum_{t \in \mathcal{T}_S} \lambda_t \left(\bigotimes_{i=1}^{J} \mathbf{E}_t^{(i)} - \bigotimes_{i=1}^{J} \mathbf{D}_t^{(i)} \right), \quad (1)$$

where $\mathbf{Q}_l^{(i)}$ is the generator matrix of local transitions in component i $(1 \leq i \leq J)$, $\mathcal{T}_S$ is the set of synchronized transitions, λ_t is the rate of synchronized transition t, $\mathbf{E}_t^{(i)}$ is the synchronized transition matrix with respect to component i and transition t, $\mathbf{D}_t^{(i)} = \mathrm{diag}(\mathbf{E}_t^{(i)}\mathbf{e}^T)$, and $\mathbf{e}$ is an n_i-dimensional vector of ones.

Let $\mathcal{RS}_i = \{0, \ldots, n_i - 1\}$ be the set of states of component i, and define $\lambda_{l_i} = \max_{x \in \mathcal{RS}_i}(|\mathbf{Q}_l^{(i)}(x,x)|)$ and $\Lambda = \sum_{i=1}^{J} \lambda_{l_i} + \sum_{t \in \mathcal{T}_S} \lambda_t$. Applying uniformization to the generator matrix of component i yields the transition matrix, $\mathbf{P}_l^{(i)} = \mathbf{Q}_l^{(i)}/\lambda_{l_i} + \mathbf{I}$, for a DTMC embedded in a Poisson process of rate $\lambda_{l_i} s$ that has density $\beta(\lambda_{l_i} s, k) = e^{-\lambda_{l_i} s}(\lambda_{l_i} s)^k / k!$.

Further, define $\mathcal{E}(t) = \{i \mid \exists\, 0 \leq x < n_i : \mathbf{e}_x \mathbf{E}_t^{(i)} \mathbf{e}^T < 1\}$ to be the set of components that may disable transition t. For each synchronized transition $t \in \mathcal{T}_S$, we define the following matrices, which correspond to elements in the set $\mathcal{E}(t)$: $\overline{\mathbf{E}}_t^{(i)} = \mathbf{I} - \mathbf{D}_t^{(i)}$.

A path in the approach consists of a sequence of component-level transitions. The generation of paths can be considered as an enumeration of strings from the alphabet $\mathcal{A} = \{l_1, \ldots, l_J\} \cup \mathcal{T}_S \cup (\cup_{t \in \mathcal{T}_S} \mathcal{E}'(t))$, where l_i denotes a local transition in component i and $\mathcal{E}'(t) = \{\bar{t}_i \mid i \in \mathcal{E}(t)\}$ is the set of artificial transitions of which $\bar{t}_i$ for component i may be generated if the rate of synchronized transition t depends on the states of i. Let $\mathcal{P}$ be the language of all strings over $\mathcal{A}$, and let $\mathcal{P}^l \subseteq \mathcal{P}$ be the language of strings of length l. For any $\boldsymbol{\pi} \in \mathcal{P}$, $\boldsymbol{\pi}(i) \in \mathcal{A}$ is the ith element and $|\boldsymbol{\pi}|$ is the length of the string $\boldsymbol{\pi}$. Thus, $\mathcal{P}$ corresponds to the set of all paths over which a model may evolve, and $\mathcal{P}^l$ contains all paths of length l. Now consider a specific path $\boldsymbol{\pi} \in \mathcal{P}^l$, and let $\mathbf{p}_0 = \otimes_{j=1}^{J} \mathbf{p}_0^{(j)}$ be the initial state distribution vector and $\mathbf{r} = \odot_{j=1}^{J} \mathbf{r}^{(j)}$ be the reward vector. Analogously, let $\mathbf{p}_0^{(i)}$ and $\mathbf{r}^{(i)}$ be the initial distribution and reward vectors for component i, and let $\odot$ denote either $\otimes$ or $\oplus$ based on whether reward is computed by a product or a sum of constituent component rewards. The reward after the transition of the path is

$$R(\boldsymbol{\pi}) = \bigotimes_{j=1}^{J} \mathbf{p}_0^{(j)} \left(\prod_{k=1}^{|\boldsymbol{\pi}|} \mathbf{P}_{\boldsymbol{\pi}(k)} \right) \bigodot_{j=1}^{J} \mathbf{r}^{(j)}, \quad (2)$$

where, for $n_j^i = \prod_{k=j}^{i} n_k$, $\mathbf{P}_{l_i} = \mathbf{I}_{n_1^{i-1}} \otimes \mathbf{P}_l^{(i)} \otimes \mathbf{I}_{n_{i+1}^J}$, $\mathbf{P}_t = \bigotimes_{i=1}^{J} \mathbf{E}_t^{(i)}$, and $\mathbf{P}_{\bar{t}_i} = \bigotimes_{j=1}^{i-1} \mathbf{D}_t^{(j)} \otimes \overline{\mathbf{E}}_t^{(i)} \otimes \mathbf{I}_{n_{i+1}^J}$

Note that each path at the component level (as shown above) corresponds to a set of trajectories at the state level. Further, observe that there are countably infinitely many paths over which a model may evolve. Thus, although a set of trajectories can be computed simultaneously, evaluating a large number of paths efficiently is still an important performance consideration. The probability of the path $\boldsymbol{\pi}$ is given by $Prob(\boldsymbol{\pi}) = \prod_{k=1}^{|\boldsymbol{\pi}|} \frac{\lambda_{\boldsymbol{\pi}(k)}}{\Lambda}$, where $\lambda_{\bar{t}_i} = \lambda_t$. The expected reward at time s and the expected accumulated reward in the interval $[0, s)$ can then be computed by

$$E[R_s] = \sum_{l=0}^{\infty} \beta(\Lambda s, l) \sum_{\boldsymbol{\pi} \in \mathcal{P}^l} Prob(\boldsymbol{\pi}) R(\boldsymbol{\pi}) \quad (3)$$

$$E[AR_s] = \sum_{l=0}^{\infty} \frac{1}{\Lambda} \left(1 - \sum_{k=0}^{l} \beta(\Lambda s, k) \right) \sum_{\boldsymbol{\pi} \in \mathcal{P}^l} Prob(\boldsymbol{\pi}) R(\boldsymbol{\pi}). \quad (4)$$

Equations (3) and (4) state that the expected rewards are just weighted sums of path rewards $R(\boldsymbol{\pi})$ for all paths $\boldsymbol{\pi} \in \mathcal{P}$. The lower and upper bounds on the finite truncations of (3) and (4) can be computed according to the derivation in [6].

2.2. Algorithm for Exploring Paths

As shown in the previous section, a model may evolve over many paths. Exploring and computing those paths efficiently is a critical consideration in achieving good performance. Since the structured path-based approach is formulated at a higher level of path abstraction, we can use a partial order reduction method to reduce considerably the number of paths that the approach has to explore directly. In particular, the approach explores only *canonical paths*. Each canonical path is a representative path of a set of equivalent paths induced by a partial order reduction relation. Once a canonical path has been explored, all of the equivalent paths can be determined from it inexpensively (see [6] for details on the path equivalence relation). In addition to exploring only canonical paths, an efficient algorithm ought to reuse as many results from the computed canonical paths as possible. The algorithms for exploring canonical paths and efficiently computing equivalent paths are described below.

Canonical paths can be explored through a definition of a lexicographical order among them. That is done through the imposition of an order among the transitions in $\mathcal{A} = \{l_1, \ldots, l_J\} \cup \mathcal{T}_S \cup (\cup_{t \in \mathcal{T}_S} \mathcal{E}'(t))$. Given two paths, $\boldsymbol{\pi}_1 = \boldsymbol{\pi} \circ a$ and $\boldsymbol{\pi}_2 = \boldsymbol{\pi} \circ b$, $a, b \in \mathcal{A}$, $\boldsymbol{\pi} \in \mathcal{P}$, then $\boldsymbol{\pi}_2$ is lexicographically greater than $\boldsymbol{\pi}_1$ if b is lexicographically greater than a according to the ordering of the transitions in $\mathcal{A}$. Thus, an algorithm for exploring canonical paths considers only the lexicographically largest path among a set of equivalent paths.

More formally, we explore canonical paths in the following way. Let $\mathcal{TS} = \mathcal{T}_S \cup (\cup_{t \in \mathcal{T}_S} \mathcal{E}'(t))$ be the set of all synchronized transitions, ε be the null path, $\mathcal{CP}$ be the set of canonical paths, and $\mathcal{CP}^l$ be the set of canonical paths of length l. Then, $\mathcal{CP}^0 = \{\varepsilon\}$ and $\mathcal{CP} = \bigcup_{l \geq 0} \mathcal{CP}^l$. For $l > 0$, the set $\mathcal{CP}^l$ can be generated lexicographically in accordance with

$$\mathcal{CP}^l = \Big\{ \boldsymbol{\pi} \circ a \mid \boldsymbol{\pi} \in \mathcal{CP}^{l-1},\ a \in \mathcal{TS} \vee \\ \big(a = l_i \wedge \big(\boldsymbol{\pi}(|\boldsymbol{\pi}|) \in \mathcal{TS} \vee \big(\boldsymbol{\pi}(|\boldsymbol{\pi}|) = l_j \wedge j \leq i\big)\big)\big) \Big\}. \tag{5}$$

Using this canonical-path exploration algorithm, we can reuse the computed results (such as the state distribution vectors) from paths that have already been explored. The algorithm uses a depth-first exploration strategy to minimize the amount of required storage by eliminating the need to store all intermediate results from the computed canonical paths.

Let $card(\boldsymbol{\pi})$ be the cardinality of the set of paths equivalent to $\boldsymbol{\pi}$. Each path $\boldsymbol{\pi} \in \mathcal{CP}$ represents $card(\boldsymbol{\pi})$ paths in $\mathcal{P}$ whose rewards, probabilities, and state distribution vectors are all identical. For the computation of $card(\boldsymbol{\pi})$, we define the functions $cs(\boldsymbol{\pi})$ and $cl^{(i)}(\boldsymbol{\pi})$, $i = 1, \ldots, J$, on paths in the following way. Let $cs(\varepsilon) = cl^{(i)}(\varepsilon) = 0$ and

$$cs(\boldsymbol{\pi} \circ a) = \begin{cases} card(\boldsymbol{\pi}) & \text{if } a \in \mathcal{TS} \vee \\ & (a \notin \mathcal{TS} \wedge \boldsymbol{\pi}(|\boldsymbol{\pi}|) \in \mathcal{TS}), \\ cs(\boldsymbol{\pi}) & \text{otherwise} \end{cases} \tag{6}$$

$$cl^{(i)}(\boldsymbol{\pi} \circ a) = \begin{cases} cl^{(i)}(\boldsymbol{\pi}) + 1 & \text{if } a = l_i \\ 0 & \text{if } a \in \mathcal{TS}. \\ cl^{(i)}(\boldsymbol{\pi}) & \text{otherwise} \end{cases} \tag{7}$$

The function $cs(\boldsymbol{\pi})$ computes the cardinality up to the last synchronized transition in $\boldsymbol{\pi}$, and $cl^{(i)}(\boldsymbol{\pi})$ counts the local transitions after the synchronized transition. Both functions together implement a partial order reduction to compute equivalent paths in which the local transitions between two immediate synchronized transitions can be arbitrarily interchanged. The algorithm uses those functions to compute $card(\boldsymbol{\pi})$ in the following way:

$$card(\boldsymbol{\pi}) = cs(\boldsymbol{\pi}) \cdot \frac{\left(\sum_{i=1}^{J} cl^{(i)}(\boldsymbol{\pi})\right)!}{\prod_{j=1}^{J} \left(cl^{(i)}(\boldsymbol{\pi})!\right)} \tag{8}$$

In this section, we described briefly how a model is solved through the computation of various attribute values at the path level. In the next section, we show how the new technique of path composition can compute those values more efficiently at the subpath level.

3. Algorithms for Path Composition and Path Selection

We now present the main technical contribution of this paper. This section is presented in the following manner. First, we give the justification for how paths can be composed from subpaths. This is done by showing how paths can be decomposed into subpaths. Next, we show how the subpath values are computed and, subsequently, how they can be used to compute the path values. We then present the complete and necessarily efficient algorithm for composing the subpaths. The last part of this section describes an approach for selecting important paths from the precomputed subpaths.

3.1. The Path-Composition Algorithm

The essence of the path-composition algorithm is that it is much quicker to compute paths by composing subpaths than to compute the paths themselves directly as in (2), (3), and (4). To that end, we explain here how they can be composed by showing how they can be decomposed into subpaths. Let l_1 and l_2 be the local transitions from components 1 and 2, $t_{\{1,2\}}$ be a synchronized transition between components 1 and 2, and $t_{\{4,5\}}$ be a synchronized transition that does not affect components 1 and 2. There are three general cases to consider:

Case 1: Given that a path consists of local transitions only, such as $l_1 \circ l_2 \circ l_1 \circ l_2$, we can decompose it into two subpaths $l_1 \circ l_1$ and $l_2 \circ l_2$, since l_1 does not affect component 2 and similarly l_2 does not affect component 1.

Case 2: Given that a path consists of affecting synchronized transitions, such as $l_1 \circ l_2 \circ t_{\{1,2\}} \circ l_1 \circ l_2$, we can decompose it into two subpaths $l_1 \circ t_{\{1,2\}} \circ l_1$ and $l_2 \circ t_{\{1,2\}} \circ l_2$, in which we compute the effect of the synchronized transition on the components individually.

Case 3: Given that a path consists of non-affecting synchronized transitions, such as $l_1 \circ l_2 \circ t_{\{4,5\}} \circ l_1 \circ l_2$, we can decompose it into two subpaths $l_1 \circ l_1$ and $l_2 \circ l_2$, since the synchronized transition has no effect on the components.

Thus, paths can be composed through identification of the sets of subpaths that make up the paths. Moreover, the subpaths can be computed individually and, thereafter, characterized by a few real values. We present the path-composition algorithm in detail in the next subsection. In the rest of this subsection, we show how the subpath values are computed.

Let us assume for the moment that all components in a model participate in all synchronizations. That assumption can be relaxed easily later without affecting the correctness of the analysis; doing so now would unduly complicate the discussion and the notations used. Define $\mathcal{A}^{(i)} = \mathcal{T}_S \bigcup (\bigcup_{t \in \mathcal{T}_S} \bar{t}_i) \bigcup \{l_i\}$ to be the set of all transitions that may affect component i. For a given path $\boldsymbol{\pi}$, let $\boldsymbol{\pi}^{(i)}$ be a basic subpath of $\boldsymbol{\pi}$ consisting only of transitions in $\mathcal{A}^{(i)}$. Thus, $\boldsymbol{\pi}^{(i)}$ is the projection of $\boldsymbol{\pi}$ onto $\mathcal{A}^{(i)}$.

As shown in [6], the state distribution at the end of a path is computed by the Kronecker product of the component vectors. As a result, we can compute the state distribution vector for component i after the transition of the subpath $\boldsymbol{\pi}^{(i)}$ by

$$\mathbf{p}^{(i)}[\boldsymbol{\pi}^{(i)}] = \mathbf{p}_0^{(i)} \prod_{k=1}^{\left|\boldsymbol{\pi}^{(i)}\right|} \boldsymbol{\Phi}_{\boldsymbol{\pi}^{(i)}(k)}, \tag{9}$$

where

$$\boldsymbol{\Phi}_{\boldsymbol{\pi}^{(i)}(k)} = \begin{cases} \mathbf{P}_l^{(i)} & \text{if } \boldsymbol{\pi}^{(i)}(k) = l_i \\ \mathbf{E}_t^{(i)} & \text{if } \boldsymbol{\pi}^{(i)}(k) = t \text{ for } t \in \mathcal{T}_S \\ \mathbf{D}_t^{(i)} & \text{if } \boldsymbol{\pi}^{(i)}(k) = \bar{t}_j \text{ for } i < j \text{ and } t \in \mathcal{T}_S \\ \overline{\mathbf{E}}_t^{(i)} & \text{if } \boldsymbol{\pi}^{(i)}(k) = \bar{t}_i \text{ for } t \in \mathcal{T}_S \\ \mathbf{I}_{n_i} & \text{otherwise} \end{cases} .$$

The state distribution vector after the transition of the path $\boldsymbol{\pi}$ is $\mathbf{p}[\boldsymbol{\pi}] = \otimes_{j=1}^{J} \mathbf{p}^{(j)}[\boldsymbol{\pi}^{(j)}]$, and from (2), the reward of the path is computed by $R[\boldsymbol{\pi}] = \left(\bigotimes_{j=1}^{J} \mathbf{p}^{(j)}[\boldsymbol{\pi}^{(j)}]\right)\left(\bigodot_{j=1}^{J} \mathbf{r}^{(j)}\right)$. Note that $\mathbf{p}^{(j)}[\boldsymbol{\pi}^{(j)}]$ is a row vector, whereas $\mathbf{r}^{(i)}$ is a column vector. If $\odot$ is $\otimes$ (i.e., rewards are computed as a product of component rewards), then $R[\boldsymbol{\pi}] = \bigotimes_{j=1}^{J} \mathbf{p}^{(j)}[\boldsymbol{\pi}^{(j)}] \bigotimes_{j=1}^{J} \mathbf{r}^{(j)} = \prod_{j=1}^{J} \mathbf{p}^{(j)}[\boldsymbol{\pi}^{(j)}]\mathbf{r}^{(j)}$. When $\odot$ is $\oplus$, the computation is slightly more complex, yielding

$$\begin{aligned} R[\boldsymbol{\pi}] &= \left(\bigotimes_{j=1}^{J} \mathbf{p}^{(j)}[\boldsymbol{\pi}^{(j)}]\right)\left(\bigoplus_{j=1}^{J} \mathbf{r}^{(j)}\right) \\ &= \sum_{j=1}^{J} \left(\bigotimes_{i=1}^{J} \mathbf{p}^{(i)}[\boldsymbol{\pi}^{(i)}]\right)\left(\mathbf{e}_{n_1^{i-1}}^{T} \otimes \mathbf{r}^{(j)} \otimes \mathbf{e}_{n_{i+1}^{J}}^{T}\right) \\ &= \sum_{j=1}^{J} \left(\mathbf{p}^{(j)}[\boldsymbol{\pi}^{(j)}]\mathbf{r}^{(j)} \left(\prod_{i=1, i\neq j}^{J} \mathbf{p}^{(i)}[\boldsymbol{\pi}^{(i)}]\mathbf{e}_{n_i}^{T}\right)\right) \end{aligned}$$

where $\mathbf{e}_n$ is a row vector of length n with all elements equal to 1. For notational convenience define $\nu^{(i)}(\boldsymbol{\pi}^{(i)}) = \mathbf{p}^{(i)}[\boldsymbol{\pi}^{(i)}]\mathbf{r}^{(i)}$ and $\xi^{(i)}(\boldsymbol{\pi}^{(i)}) = \prod_{j=1, i\neq j}^{J} \mathbf{p}^{(j)}[\pi^{(j)}]\mathbf{e}_{n_j}^{T}$ such that the reward for the path $\boldsymbol{\pi}$ can be computed as either

$$R[\boldsymbol{\pi}] = \prod_{i=1}^{J} \nu^{(i)}(\boldsymbol{\pi}^{(i)}) \;\text{ or }\; \sum_{i=1}^{J} \nu^{(i)}(\boldsymbol{\pi}^{(i)}) \cdot \xi^{(i)}(\boldsymbol{\pi}^{(i)}) . \tag{10}$$

Similarly, the probability of a path can be decomposed into the subpath probabilities due to the local and synchronized transitions. The probability of the subpath due to the local transitions is $ProbL(\boldsymbol{\pi}^{(i)}) = \prod_{k \in \{1,\ldots,\left|\boldsymbol{\pi}^{(i)}\right|\} \wedge \boldsymbol{\pi}^{(i)}(k) = l_i} \frac{\lambda_{l_i}}{\Lambda}$, and the probability due to the synchronized transitions is $ProbS(\boldsymbol{\pi}) = \prod_{k \in \{1,\ldots,|\boldsymbol{\pi}|\} \wedge \boldsymbol{\pi}(k) \in \mathcal{TS}} \frac{\lambda_{\boldsymbol{\pi}(k)}}{\Lambda}$, such that

$$Prob(\boldsymbol{\pi}) = ProbS(\boldsymbol{\pi}) \prod_{i=1}^{J} ProbL(\boldsymbol{\pi}^{(i)}). \tag{11}$$

Thus, we can exploit redundant computation across many paths to improve performance by first decomposing the paths into basic subpaths. Various real values of the subpaths are then precomputed and cached. Afterward, the subpaths can be composed, and their cached values can be used to compute the bounds on the rewards computed by (3) and (4).

Before we introduce the algorithm for composing the subpaths, we present a small example to show the advantages of an isolated computation and caching of $\nu^{(i)}(\boldsymbol{\pi}^{(i)})$ and $\xi^{(i)}(\boldsymbol{\pi}^{(i)})$. A model that has two components and a single synchronized transition t that can be disabled only by the first component will be analyzed. Thus, given $\mathcal{A} = \{l_1, l_2, t, \bar{t}_1\}$, we consider the set of paths of length 3, and we assume that each matrix $\mathbf{P}_l^{(i)}$ contains nz non-zeros and that the remaining matrices contain $nz/2$ elements such that the effort it takes to analyze a path of length x equals $x \cdot nz$. $\mathcal{P}^3$ contains $4^3 = 64$ different paths, and since each path requires a computational effort of $3nz$, the overall cost for this naive computation is on the order of $192nz$.

Observe that the number of paths at the state level is usually much larger and that every path in our approach usually represents a large number of paths at the state level. By storing intermediate vectors $\mathbf{p}^{(i)}[\boldsymbol{\pi}^{(i)}]$, one can avoid having to recompute the vectors many times. Thus, if path $l_1 l_1 l_1$ has been analyzed, path $l_1 l_1 l_2$ can be computed with a cost of only nz. From a computational point of view, the computation requires a depth-first traversal of the tree that describes all possible paths. The number of vectors to be stored equals the depth of the tree, which is acceptable since the vectors have dimensions that correspond to the sizes of the component state spaces and not the overall state space. Because already-computed vectors are reused, the number of required operations is proportional to $81nz$. Exploitation of the partial order reduction reduces the number of paths from 64 to 52. The effect of the reduction is relatively small for this example, since the path length is short and we consider only two components. Thus, only the sequences of local transitions can be reordered; for example, $l_1 l_1 l_2$, $l_1 l_2 l_1$, and $l_2 l_1 l_1$ are equivalent paths, and the first is the canonical path. If intermediate results are reused, the cost of the path computation using the canonical paths is proportional to $76nz$.

If we consider local paths for the components, then the first component describes $3^3 = 27$ paths. For the estimation of the computational cost, recall that at the component level, the multiplication of a vector with a matrix describing a synchronization requires an effort in $0.5nz$ rather than nz. Since there is 1 path with 3 local transitions, there are 6 paths with 2 local transitions, there are 12 paths with 1 local transition, and there are 8 paths without local transitions, the total cost is proportional to $54nz$. Additionally, since vectors at the component level can be reused, the cost can be reduced to $26nz$. For the second component, we have only two different transitions, because the synchronized transition can be disabled only by the first component. Thus, there are 8 different paths, and the computational cost is proportional to $18nz$ without reuse of intermediate vectors and is proportional to $10.5nz$ with reuse of the vectors. If we reuse vectors, then the overall cost is proportional to $36.5nz$. Afterward, all values $\nu^{(i)}(\boldsymbol{\pi}^{(i)})$ and $\xi^{(i)}(\boldsymbol{\pi}^{(i)})$ are known for local paths with lengths up to 3. The values can be used in (10) for the computation of the reward values.

Observe that using the local paths with lengths up to 3, it is possible to compute rewards for all global paths with lengths less than or equal to 3 and also some other global paths with lengths 4, 5, and 6. Computing those additional global paths of lengths greater than 3 is inexpensive, and it helps to tighten the lower bounds further. To provide a fair comparison with our previous work, however, our implementation of the path decomposition algorithm does not take advantage of the additional global paths. The results we present in Section 4 show the performance comparison up to the same path lengths for both implementations.

In this small example, the effect of local path computation is a reduction of the computational effort by a factor of 2. However, with an increasing path length, an increasing number of components, and an increasing number of synchronized transitions, the effect grows exponentially. In the following section, we introduce an algorithm for efficient computation of rewards for global paths from the results of local subpaths.

3.2. Algorithms for Efficient Composition of Subpaths

We have shown how paths can be decomposed into subpaths and how the subpath values are computed. In order to gain the benefits of the path-composition algorithm, we must have an effective strategy for exploring the subpaths and composing them efficiently. There are several strategies for exploring and computing the basic subpaths. Either they may all be precomputed before any computation of reward bounds begins, or they may be computed when required during the computation of the bounds. In both cases, the storage complexity is combinatorial in the number of subpaths. With so many paths to consider, the selection of the valid subpaths to compose is also expensive. We now describe an efficient algorithm for precomputing and composing the subpaths.

Instead of computing and storing all of the subpaths en masse, we can compute and store sets of them in stages. At each stage, we identify a set of subpaths that are *composable* with each other and compute only those. Subpaths are composable when they have the same longest common subsequence (LCS) [3] of synchronized transitions. In order to simplify the computation, we assume that the synchronized transitions cannot be transposed past each other, so that the LCS is unique. Thus, we can partition the set of all subpaths into classes of composable subpaths, with each class characterized by a sequence of synchronized transitions. For example, suppose $t_1, t_2, t_3 \in \mathcal{TS}$ are synchronized transitions. Then, the class of composable subpaths that have the sequence $\langle t_1, t_2, t_3 \rangle$ is $\{{l_j}^* \circ t_1 \circ {l_j}^* \circ t_2 \circ {l_j}^* \circ t_3 \circ {l_j}^*\}$, where ${l_j}^*$ denotes zero or more occurrences of the local transition of component j $(1 \le j \le J)$.

More formally, given a sequence of synchronized transitions $\langle t_1, \ldots, t_n \rangle$ and a maximum subpath length $L = n + K$, the class of composable subpaths corresponding to the sequence is

$$\mathcal{CS}(\langle t_1, \ldots, t_n \rangle, L) = \bigcup_{j=1}^{J} \bigcup_{k_0^{(j)} + \ldots + k_n^{(j)} = 0}^{K} \left\{ \boldsymbol{\pi}^{(j)} = {l_j}^{k_0^{(j)}} \circ t_1 \circ {l_j}^{k_1^{(j)}} \circ t_2 \circ {l_j}^{k_2^{(j)}} \circ \ldots \circ t_n \circ {l_j}^{k_n^{(j)}} \right\}. \tag{12}$$

For each subpath $\boldsymbol{\pi}^{(j)} \in \mathcal{CS}(\langle t_1, \ldots, t_n \rangle, L)$, we need to precompute and store only one or two real values, namely $\nu^{(i)}(\boldsymbol{\pi}^{(j)})$ and $\xi^{(i)}(\boldsymbol{\pi}^{(j)})$. The probability of the subpath is given by

$$Prob^{(j)}(K) = \frac{\lambda_{l_j}^K}{\Lambda^K}, \tag{13}$$

and the probability of the synchronized transitions equals

$$ProbS(t_1 \ldots t_n) = \frac{\prod_{k=1}^{n} \lambda_{t_k}}{\Lambda^n}. \tag{14}$$

Each path that is composed from some subpaths in the class $\mathcal{CS}(\langle t_1, \ldots, t_n \rangle, L)$ can be considered as a canonical path that represents a set of equivalent paths. Computing the number of equivalent paths using the new algorithm is now straightforward (as compared to (8) in Section 2.2). The number of equivalent paths is simply

$$\prod_{i=0}^{|\langle t_1, \ldots, t_n \rangle|} \frac{\left(\sum_{j=1}^{J} k_i^{(j)}\right)!}{\prod_{j=1}^{J} \left(k_i^{(j)}!\right)}, \tag{15}$$

which follows from the number of possible reorderings of the local transitions between two immediate synchronized transitions.

Using (10), (13), (14), and (15), we can now compute the expected instantaneous reward for a model. Let Ψ be the set of all sequences of synchronized transitions. Then, the expected instantaneous reward is computed by

$$E[R_s] = \sum_{\psi \in \Psi} \sum_{m=|\psi|}^{\infty} \beta(\Lambda s, m) \sum_{\sum_{j=1}^{J} \sum_{i=0}^{|\psi|} k_i^{(j)} + |\psi| = m} \Bigg\{ \prod_{i=0}^{|\psi|} \frac{\left(\sum_{j=1}^{J} k_i^{(j)}\right)!}{\prod_{j=1}^{J} \left(k_i^{(j)}!\right)} \cdot ProbS(\psi) \cdot \prod_{j=1}^{J} Prob^{(j)} \left(\sum_{i=0}^{|\psi|} k_i^{(j)} \right) \cdot \prod_{j=1}^{J} \nu^{(j)} \left(l_j^{k_0^{(j)}}, \psi_1, \ldots, \psi_{|\psi|}, l_j^{k_{|\psi|}^{(j)}} \right) \Bigg\} \tag{16}$$

when the reward is computed from the product of component rewards. For an additive reward, we must use the summation of (10) instead of the last product in (16). The equations for the expected accumulated reward can be derived in a similar manner.

In summary, the algorithm works by generating the set of sequences of synchronized transitions. For each sequence, a class of composable subpaths is explored, and their values are precomputed using (10) and (13). Afterward, the subpaths are composed, and their values are used to compute the expected reward of a model. The subpaths in a class of composable subpaths may be explored using a depth-first strategy similar to that discussed in Section 2.2 to minimize the amount of memory used to store intermediate results. The algorithm is storage-efficient because during each stage of the computation, only two real values for each subpath are stored. Moreover, it is computationally efficient because each subpath, as a redundant computation across many paths, is computed only once and reused many times.

3.3. An Approach for Selecting Important Subpaths

Although there are many paths to consider, a large number of them often contribute little or no reward toward the computation of the bounds on the solution of a model. We can thus speed up the computation further by identifying important paths and discarding those that contribute little or nothing toward tightening of the bounds. While path selection has been considered by other researchers, our approach is new in that it bases the selection on additional information available from the computed subpaths.

There are several subpath factors that directly affect the reward contribution of a path. With respect to (16), one of the main factors is the subpath reward $\nu^{(i)}(\boldsymbol{\pi}^{(i)})$. If a subpath has zero reward, all paths composed from it also have zero reward. Using this insight, we can improve the performance of the computation if we can identify efficiently those subpaths that contribute no reward and discard them from further computation.

Starting from (9) and (10), we note that $\nu^{(i)}(\boldsymbol{\pi}^{(i)})$ can be computed efficiently by first computing the projection of a component reward vector $\mathbf{r}^{(i)}$ onto component i. That yields a projected reward vector that can be cached and reused repeatedly. Next, when subpaths are being explored, we can compute their reward values efficiently by means of a scalar product of the subpath state distribution vector and the projected reward vector. In complexity terms, that incurs a cost of $\mathcal{O}(n_i)$ rather than $\mathcal{O}(n_i^2)$, where n_i is the size of the state space of component i.

We implement the approach by computing all projected reward vectors and caching them before any subpath is computed. As the subpaths are explored, the projected reward vectors are used to compute the subpath reward values. Those subpaths that contribute non-zero reward values are retained for composition with other subpaths; the rest are discarded immediately. When a zero-reward subpath is discarded, we keep the implementation simple by also discarding the successive subpaths that can be generated from the zero-reward subpath. Though the discarded successive subpaths may have non-zero reward values, their contributions toward tightening the bounds appear to be negligible in our experiments. In the next section, we show example results that are obtained using this approach.

4. Numerical Results

We evaluate our algorithm by studying its performance in analyzing two models with very different characteristics: a model of a distributed information service system adapted from the model in [8] and a model of a media multicast system inspired by the work of Chu et al. [2]. In the former model, we evaluate the reliability and availability of the system; in the latter model, we evaluate the performability properties of the corresponding system. Moreover, the latter model has more components, a larger state space, and tighter coupling among the components. After describing the systems in detail below, we present the performance results and compare them with the results obtained using our earlier approach described in [6].

4.1. Model Description of the Distributed Information Service System

We augment the original model of the distributed information service system with synchronized transitions among the components to describe how faults are propagated through the system. In addition, we increase the number of front-end modules in order to model the occurrence of a fault only when a majority of the modules are corrupted. We also model double redundancy in the processing units by adding an additional module for every module in the original processing units. These additions quickly increase the size of our model, resulting in a model with approximately 2.7×10^{18} states, so large that it could not have been analyzed using traditional techniques, but can be using our approach.

The model consists of six front-end modules that interact with four processing units. Each processing unit consists of redundant components, including two processors, two switches, two memory units, and two databases. Each of the components has its own repair facility. All of them go through the cycle of *Working*, *Corrupted*, *Failed*, and *Repaired*. The *stochastic activity network* (SAN) model of the system is shown in Fig. 1.

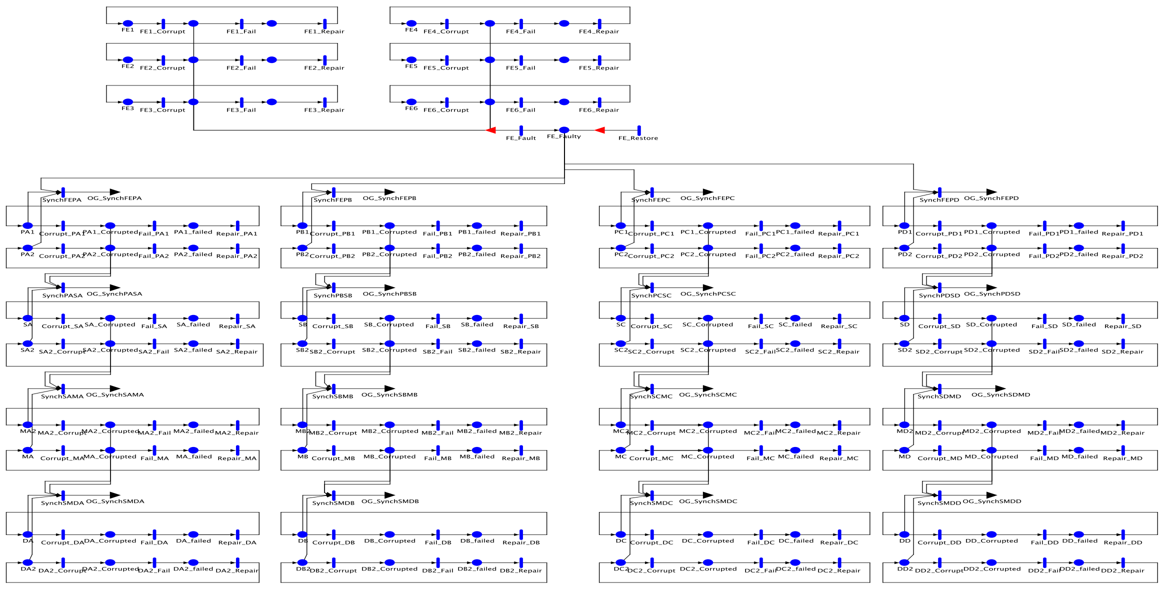


Figure 1. SAN model of a distributed information service system.

Fault propagation in the system is modeled as follows:

- When a majority of the six front-end modules are corrupted, the front-end is considered faulty, and it may propagate the error to any of the four processing units in which there are two working processors. Propagation occurs via the synchronized activities between the front-end and the processors in the processing units. The front-end or any of the processors may disable the synchronized activities. After propagating the error to a processing unit, the front-end may remain in the faulty state and continue to propagate errors to other processing units until the majority fails or are repaired or there are no more processing units to which the error can be propagated.

- When both processors in a processing unit are corrupted, they both may propagate the error to their working switches via a synchronized activity. Any of the involved components may disable the synchronized activity. After the error propagation, the processors may remain in the corrupted state until they fail.

- When both switches of a processing unit are corrupted, they may propagate their errors to the working memory units via a synchronized activity. Any of these components may disable the activity. After propagating the error, the switches may remain in the corrupted state until they fail.

- When both memory units of a processing unit are corrupted, they may propagate their errors to the working databases via a synchronized activity. Any of these components may disable the activity. After propagating the error, the memory units may remain in the corrupted state until they fail.

We vary activity rates in the submodels and among the synchronized activities, so the resulting model does not have symmetries that would allow it to be lumped. Because of space constraints, we do not list the rates used for the model here. In total, the model has 5 submodels (modeling 38 components) and 4 synchronized activities. Because each component has three states, the state space of the whole model has $2 \times 3^{38} \approx 2.7 \times 10^{18}$ states. We computed the reliability of the system over the interval $[0, 1.0]$, the point availability at time 0.1, and the interval availability over the interval $[0.0, 0.1]$ when all components in the model were in the working state.

4.2. Model Description of the Media Multicast System

The SAN model of the media multicast system is shown in Fig. 2. The model is parameterized by many variables, such that by varying the parameters for the activity rates and buffer sizes, we can measure the sensitivity of the system and the likelihood that it will experience buffer overflow. In addition, we can also compute the probability of having to flush the frame buffers when the system is corrupted. The model consists of a source (CMU) that multicasts frames to the clients Berkeley, UIUC1, and UWisc. Berkeley and UIUC1, in turn, multicast the frames further to UCSB, UIUC2, and UKY. Note that UWisc synchronizes only with CMU; Berkeley is more tightly connected with CMU and UCSB; and UIUC1 must synchronize with CMU, UIUC2, and UKY. The complete model has seven submodels and approximately 1.9×10^{20} states.

Frames are initially generated by the source, CMU. The tasks of the clients are to decode the frames, process them (perhaps adding more information to them), and encode them for further multicast. All of these components may be in any

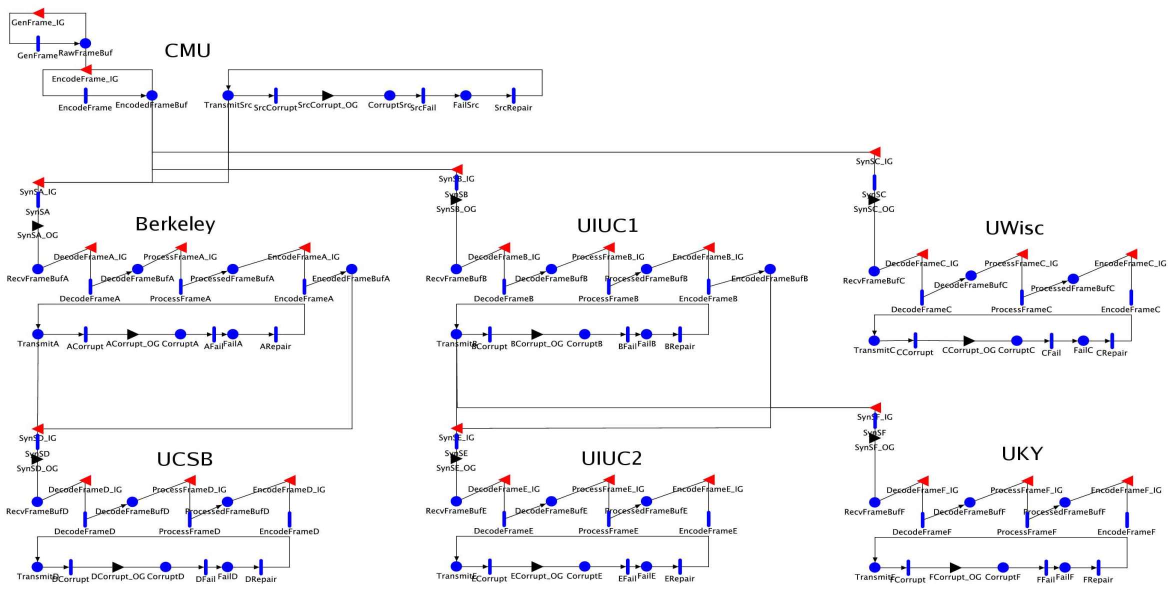


Figure 2. SAN model of a media multicast system.

one of the operational, corrupted, or failed modes at any instant of time, and they all have their own repair facilities. They may transmit frames only when they are operational. When they are corrupted, their frame buffers are flushed, because the stored frames are presumably corrupted also. The transmitted frames are dropped when the clients' buffers are full.

Thus, the sensitivity of the system depends on the buffer sizes, the transmission rates, and the processing rates of the components. By varying these parameters, we can compute the probability of having a buffer overflow or buffer flushing at some time after the system has been in operation. In the next section, we present numerical results from our experimental evaluation. The results are not meant to be representative of any real system, since the parameters we used were not taken from a real system. They do show, however, that the parameters are interdependent and that our algorithm works correctly in computing the results for the varied parameters.

4.3. Experimental Evaluation

We conducted all of our experiments on a workstation that had the AMD Athlon XP 2700+ processor running at 2.17 GHz with 1.0 GB of RAM. The operating system was Red Hat Linux 9.0 with mounted file systems. We compiled our implementation using the compiler g++ 3.3 with optimization flag -O3 only.

For the model of the distributed information service system, the Möbius simulation results are $0.99883 \pm 2.11883414 \times 10^{-4}$ for the point availability, $0.099934 \pm 1.340263 \times 10^{-5}$ for the interval availability, and $0.98616 \pm 7.241023 \times 10^{-4}$ for reliability. These results were obtained at a 95% confidence level.

For the model of the media multicast system, the Möbius simulation results are $0.925 \pm 1.63 \times 10^{-2}$ for the probability that the system will experience a buffer overflow and $0.365 \pm 2.99 \times 10^{-2}$ for the probability that the system will have to flush its frame buffers due to a system failure. These results were also obtained at a 95% confidence level.

Tables 1 and 2 show the results for the availability and reliability measures, respectively, of the model of the distributed information service system calculated using our path-based approach. Note that the lower and upper bounds for each measure converge as the path length increases, because more paths are computed. Although the path-selection approach discards zero-reward subpaths and their successive subpaths, the bound values for this particular model are not affected up to the seventh significant digits in any of the experiments we performed. Column *Basic Algorithm Time (sec)* lists the time taken to evaluate the model through the use of the path-based approach described in [6]. Column *Path Decomposition Enhanced Algorithm Time (sec)* lists the time taken using of our new algorithm, which makes use of the path-decomposition and path-selection schemes described in this paper. As shown in the time columns for both the availability and reliability results, our new algorithm achieves approximately 80% performance improvement relative to the previous algorithm. As the path length gets longer, the algorithm performs better. For example, at the path length of 6 for the availability results, it achieves almost 85% improvement; at the same path length for the reliability results, it achieves 87% improvement. We do not have the timing result for the basic approach at the path length of 7, because it takes too long to compute.

Tables 3 and 4 show the results for the probabilities of buffer overflow and buffer flushing, respectively, for the model of the media multicast system. For this model, the values of the bounds computed by the basic approach and by the new algorithm differ somewhat. We list the values of both bounds in the tables for comparison. In order to understand better the rates of convergence of the bounds, we provide a

Table 1. Numerical results for availability

Path Length	Lower Bound Point Availability	Upper Bound Point Availability	Lower Bound Interval Availability	Upper Bound Interval Availability	Basic Algorithm [6] Time (sec)	Path Decomposition Enhanced Algorithm Time (sec)
1	2.519650e-01	9.998975e-01	6.261971e-02	9.998443e-02	1.62	0.01
2	4.978311e-01	9.996229e-01	8.131310e-02	9.996355e-02	1.88	0.08
3	7.174581e-01	9.992548e-01	9.180509e-02	9.994597e-02	4.88	0.87
4	8.645992e-01	9.989260e-01	9.680362e-02	9.993480e-02	42.25	8.88
5	9.434621e-01	9.987056e-01	9.885818e-02	9.992906e-02	504.81	92.18
6	9.786854e-01	9.985875e-01	9.959795e-02	9.992657e-02	6135.23	934.25
7	9.921700e-01	9.985347e-01	9.983439e-02	9.992565e-02	—	9673.85

Table 2. Numerical results for reliability

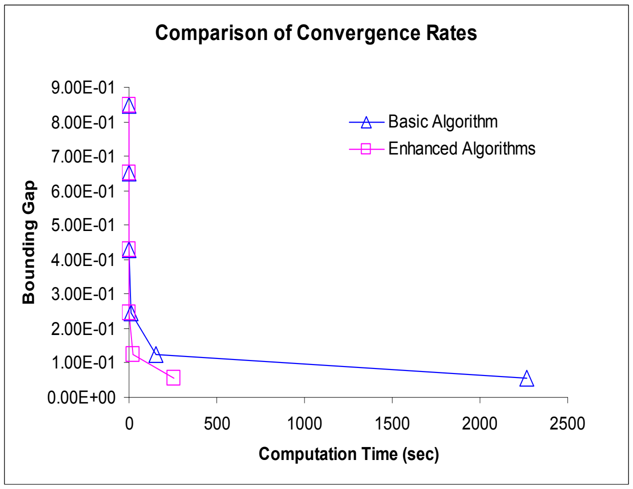
Path Length	Lower Bound Reliability	Upper Bound Reliability	Basic Algorithm [6] Time (sec)	Path Decomposition Enhanced Algorithm Time (sec)
1	6.522987e-01	9.955666e-01	1.26	0.00
2	8.661349e-01	9.902090e-01	1.52	0.08
3	9.517685e-01	9.869717e-01	4.41	0.72
4	9.774886e-01	9.856676e-01	40.38	7.30
5	9.836687e-01	9.852736e-01	487.09	74.02
6	9.849062e-01	9.851784e-01	5990.44	759.56
7	9.851185e-01	9.851592e-01	—	7763.17

Table 3. Numerical results for the probability of buffer overflow

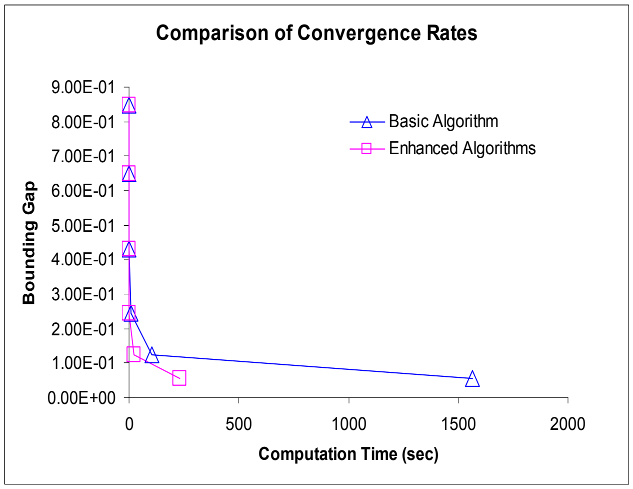
Path Length	Basic Algorithm Lower Bound	Basic Algorithm Upper Bound	Enhanced Algorithm Lower Bound	Enhanced Algorithm Upper Bound	Basic Algorithm [6] Time (sec)	Path Decomposition Enhanced Algorithm Time (sec)
1	1.492972e-01	9.962873e-01	1.492972e-01	9.962873e-01	0.21	0.00
2	3.341998e-01	9.840578e-01	3.341998e-01	9.840578e-01	0.25	0.02
3	5.345312e-01	9.644779e-01	5.339680e-01	9.639147e-01	0.89	0.19
4	6.978060e-01	9.437606e-01	6.958400e-01	9.417942e-01	10.42	2.05
5	8.045237e-01	9.273265e-01	8.007710e-01	9.235738e-01	152.80	22.71
6	8.627557e-01	9.168673e-01	8.574547e-01	9.115663e-01	2266.00	257.59

Table 4. Numerical results for the probability of buffer flushing

Path Length	Basic Algorithm Lower Bound	Basic Algorithm Upper Bound	Enhanced Algorithm Lower Bound	Enhanced Algorithm Upper Bound	Basic Algorithm [6] Time (sec)	Path Decomposition Enhanced Algorithm Time (sec)
1	1.177836e-01	9.647736e-01	1.177836e-01	9.647736e-01	0.15	0.00
2	2.146505e-01	8.645084e-01	2.148119e-01	8.646698e-01	0.18	0.02
3	2.903990e-01	7.203457e-01	2.909466e-01	7.208933e-01	0.62	0.17
4	3.348248e-01	5.807794e-01	3.358348e-01	5.817894e-01	7.09	1.89
5	3.556691e-01	4.784719e-01	3.570485e-01	4.798513e-01	103.90	20.74
6	3.638190e-01	4.179306e-01	3.654202e-01	4.195318e-01	1564.00	229.20



(a) Buffer Overflow



(b) Buffer Flushing

Figure 3. Comparison of convergence rates between the basic algorithm and the enhanced algorithm.

graphical comparison of the convergence rates in Fig. 3.

5. Summary

Even though existing path-based techniques have been shown to be effective in reducing the amount of memory necessary to analyze very large models, they are still limited in the size of problems they can solve, due to the large number of paths that often need to be explored to obtain tight bounds on a measure. This paper presented a novel approach for computing paths based on the idea of path composition. Instead of computing paths directly, the approach first computes possible subpaths for each component of a model. The sets of subpaths are then composed to explore many paths simultaneously. Effectively, this approach eliminates redundant computation expended in computing common subpaths found across multiple paths. Furthermore, we showed how a path-selection approach works seamlessly with the path-composition algorithm to find important subpaths efficiently. As a result, we were able to achieve a speedup of 6.6 to 8.8 times for two benchmark models. To the best of our knowledge, our work is the first to propose the use of path composition for the analysis of Markov models. These improvements make it feasible to evaluate efficiently models of practical systems that are significantly larger than could previously be handled.

Acknowledgment

This material is based upon work supported by Pioneer Hi-Bred and by the National Science Foundation under Grant No. 0406351. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors. The research of Peter Buchholz has been partially supported by the Deutsche Forschungsgemeinschaft as part of the Collaborative Research Center SFB 559. We would also like to thank Jenny Applequist for her editorial assistance and Tod Courtney for his help with the implementation of the algorithms.

References

[1] P. Buchholz. Exact and ordinary lumpability in finite Markov chains. *Journal of Applied Probability*, 31:59–75, 1994.

[2] Y. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *Proceedings of the ACM SIGMETRICS 2000 International Conference on Measurements and Modeling of Computer Systems*, pages 1–12, 2000.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[4] J. G. Kemeny and J. L. Snell. *Finite Markov Chains*. D. Van Nostrand Company, 1960.

[5] P. Kemper. Transient analysis of superposed GSPNs. *IEEE Transactions on Software Engineering*, 25(2):182–193, 1999.

[6] V. V. Lam, P. Buchholz, and W. H. Sanders. A structured path-based approach for computing transient rewards of large CTMCs. In *Proceedings of the First International Conference on the Quantitative Evaluation of Systems (QEST 2004)*, pages 136–145, Enschede, The Netherlands, 2004.

[7] A. S. Miner. Computing response time distributions using stochastic Petri nets and matrix diagrams. In *Proc. 13th Int. Workshop on Petri Nets and Performance Models (PNPM'03)*, pages 10–19, 2003.

[8] R. R. Muntz and J. Lui. Computing bounds on steady-state availability of repairable computer systems. *Journal of the ACM*, 41(4):676–707, 1994.

[9] D. M. Nicol and D. L. Palumbo. Reliability analysis of complex models using SURE bounds. *IEEE Transactions on Reliability*, 44(1):46–53, March 1995.

[10] M. A. Qureshi. *Construction and Solution of Markov Reward Models*. PhD thesis, University of Arizona, 1996.

[11] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.